4 Язык командного интерпретатора shell

4.1 Shell как язык программирования

Shell рассматривается как интерпретатор специализированного языка. Это же название «shell» закрепилось и за самим языком. Соответственно, существуют разные варианты языка shell, свойственные разным командным интерпретаторам (**`sh`**, **`bash`**, **`csh`** и т.д.)

В качестве языка программирования shell обеспечивает решение специализированной задачи: выполнение других программ («задач») и управление ими.

Особенность: «программа» может выполняться как из готового файла (файл сценария, пакетный режим), так и из потока – в виде отдельных вводимых оператором строк (интерактивный режим) или после перенаправления.

Аналогии с «обычными» (универсальными) языками программирования:

– алфавит

– синтаксис

– ключевые слова

– типы данных, переменные и константы

– управляющие конструкции – операторы, ключевые слова

– стандартная библиотека – операторы и встроенные команды

– подключаемые библиотеки – все исполняемые файлы в файловой системе

4.2 Алфавит

Символы и метасимволы, включая экранирующие символы (кавычка, двойные кавычки, обратный слэш).

Метасимволы: «****», «**'**», «**"**», «**#**», «**|**», «**&**», «**;**», «**(**», «**)**», «**{**», «**}**», «**[**», «**]**», «**<**», «**>**», «*****», «**?**», «**`**», « » (символ «пробел»), «**\t**» (символ «табуляция»), «**\n**» (символ «конец строки», NL).

Перечень может быть неполон. Интерпретация зависит от контекста (синтаксических конструкций).

Существенная проблема – неоднозначность интерпретации многих метасимволов, зависимость от контекста, конфликт множеств метасимволов в языках различных программ. Острота проблемы для командного интерпретатора связана с тем, что он должен работать с другими программами и, следовательно, исходными текстами на их языках.

Часть специальных символов рассмотрены в предыдущем разделе.

Символ «****» – экран или «переключатель» специального смысла одного символа

Символы «**'**…**'**» и «**"**…**"**» – экранирование символов в строке

Символы « », «**\t**», «**\n**» (если он встретился в строке) – разделители подстрок

Символ «**#**» – признак комментария (действует до конца текущей строки)

Символы «**?**», «*****», «**[**…**]**» – «подстановочные» (действуют не во всех контекстах), фактически ограниченное подмножество языка ***регулярных выражений*** (***regexp***)

Ряд символов, которые считаются «непечатными», записываются с помощью экрана «\» и имеют специальный смысл для других программ или устройств (например, для терминала), но не являются метасимволами для shell, и «экранирование» служит лишь как способ их записи.

Также «\» интерпретируется как признак восьмеричного числа.

Управляющие операторы – «разделители» командной строки (***списка***) на ***конвейеры***

Безусловное следование – команды (конвейеры) выполняются последовательно, от первого до последнего:

команды1 ; команды2

Условное следование – «выполнить если TRUE» – следующий конвейер (команда) выполняется, если код завершения предыдущего «успешный» (т.е. =0):

команды1 && команды2

Условное следование – «выполнить если FALSE» – следующий конвейер (команда) выполняется, если код завершения предыдущего «неуспешный» (т.е. ≠0):

команды1 || команды2

В любом случае итоговый код завершения командной строки (списка) – код завершения последнего выполненного конвейера (команды).

Безусловное следование бывает полезно для более компактной записи команд, в т.ч. в сложных конструкциях.
Условное следование позволяет в ряде случаев избежать гораздо более громоздких конструкций ветвления.

Пример – вывод содержимого файла без сообщений об ошибке в случае его отсутствия или недоступности:

```
test -r myfile && cat myfile
```

Оператор ***туннелирования*** (конвейеризации) – разделитель конвейера на отдельные команды:

ком1 **|** ***ком2***	туннелирование (pipe) – стандартный поток вывода (**stdout**) команды или конвейера ***ком1*** в стандартный поток ввода (**stdin**) ***ком2***
|&	??

Перенаправление ввода-вывода с точки зрения работы с потоками и файлами похоже на туннелирование, но не делит конвейер на команды, а меняет назначение потоков ввода-вывода отдельных команд (в т.ч. составных, а также вызываемых скриптов).
Если для команды стандартный поток был перенаправлен, то это переназначение остается в силе и при дальнейшем туннелировании.

Операторы ***перенаправления***:

… **>*файл***	поток вывода команды в файл, файл перезаписывается
… **>&*n***	поток вывода команды в открытый файл или поток по его дескриптору **fd=*n***
… **>>*файл***	поток вывода команды в файл, файл дописывается
… **<*файл***	поток ввода команды из файла
… ***m*>&*n***	открытый поток или файл **fd=*m*** в другой поток или файл **fd=*n***
n*<>*файл	открыть файл (для ввода и вывода) и связать с дескриптором **fd=*n***
<&-, >&-	закрыть стандартный поток ввода, поток вывода
***n*<&-, *n*>&-**	закрыть поток ввода или поток вывода с **fd=*n***

Также следует отнести к управляющим:

. (точка) – фактически встроенная команда shell – выполнение следующей за ней части командной строки как отдельной завершенной командной строки в текущем экземпляре shell, без создания новой копии (актуально при выполнении внешних скриптов). Например:

```
. ./myscript.sh
```

Здесь первая «**.**» – команда, вторая «**.**» – часть относительного пути, указывающая на текущий директорий.

& (в конце строки) – выполнять в ***фоновом*** режиме. Полезно для программ и скриптов, требующих много времени или постоянного присутствия в памяти. Например:

```
myprog_longtime &
```

Для фоновых задач сообщается PID соответствующего процесса. Фоновый процесс может быть сделан активным (перемещен на «передний план» командой **`fg`** («foreground»).

`...` (обратные кавычки) – выполнить заключенную в них команду, конвейер или список и подставить в виде строки содержимое результирующего потока вывода (stdout). Например, присваивание переменной (см. ниже) текущей даты:

```
current_date= `date`
```

Здесь критично отсутствие разделителя (пробела) перед знаком «=». В противном случае ***`current_date`*** будет интерпретироваться (скорее всего, ошибочно) как команда.

Альтернативный синтаксис:

```
current_date=$(date)
```

Здесь критично отсутствие пробелов и до, и после знака «=».

4.3 Типы данных

Основной тип – ***строка***.
В некоторых случаях (некоторыми командами) строка интерпретируется как число.
Некоторые интерпретаторы предусматривают «настоящие» числовые типы (например **`csh`**). Как правило, ограничиваются целыми числами с фиксированной разрядностью.

Массивы – поддерживаются некоторыми интерпретаторами (в том числе **bash**), но не **sh**.

Объявление массива (bash):

***имя_массива*=(*список_значений*)**

Обращение к элементу массива:

${*имя_массива*[*индекс*]}

В качестве индекса могут выступать константы и переменные, а также специальное значение «@» – обращение ко всему списку значений.

Пример:

```
A=( 1 2 3 ”a” ”b” ”c” )
echo ${A[1]}
echo ${A[$i]}
echo ${A[@]}
```

Списки – сами по себе не являются отдельным типом. Фактически это подстроки, разделенные символами-разделителями, которые можно последовательно выбирать из потока.

Пример: результат перехвата данных, выводимых командой:

```
file_list= `ls`
```

4.4 Константы

Значения, записанные непосредственно в тексте программы, в данном случае строковые – часть строки, отделенная от других символами-разделителями и другими метасимволами.

Экранирование позволяет «собрать» единый литерал из нескольких подстрок независимо от наличия символов-разделителей (а также улучшает читаемость).

Одинарные кавычки – полное экранирование специальных символов внутри строки:

```
'Hello world'
```

Двойные кавычки – то же, но с подстановкой значений переменных (см. ниже):

```
"Home directory is $HOME"
```

4.5 Переменные

Объявление (создание) переменной:

имя=”значение”

Отсутствие разделителя перед знаком «=» важно! В противном случае имя переменной интерпретируется как команда, а остальная часть строки – как ее аргументы.
Имена переменных регистрочувствительные.

Обращение к содержимому переменной:

… **$*имя*** …

Например:

```
X=123 ; y=”567” ; Z=’890’
echo $X $y $Z
```

(Здесь кавычки не важны, т.к. в строках нет разделителей)

Переменная может хранить список значений:

```
file_list=`ls`
echo $file_list
```

Переменная, значение которой не задано при объявлении, получает пустое значение:

Z= *или* **Z=""**

При попытке использования не объявленной заранее переменной также будет подставлено пустое значение, ошибка произойдет только если пустое значение недопустимо в конкретном контексте.

Различить пустые и необъявленные переменные можно с помощью соответствующих опций команды **test** (см. ниже)

Созданные переменные включаются в ***окружение*** интерпретатора (часто говорят «окружение программы, скрипта»). Каждый процесс получает при создании блок окружения и в дальнейшем может пополнять его новыми переменными, которые становятся равноправны с унаследованными (кроме передачи порожденным процессам).

Таким образом, блок окружения – список именованных переменных, словарь, список пар вида:

<*имя*> – <*значение*>

Ряд имен переменных и их содержимое закреплены в спецификациях и одинаковы во всех (или большинстве) Unix-систем.

Некоторые традиционно используемые переменные окружения:

Имя	Использование
$USER	Имя текущего пользователя
$UID, **$EUID**	Идентификаторы текущего пользователя
$CPU	Тип процессора
$HOME	«Домашний» директорий текущего пользователя
$PWD	Текущий директорий
$HOST	Имя компьютера (узла)
$LANG	Действующий в системе язык
$PATH	Список путей для поиска исполняемых файлов
$SHELL	«Оболочка» (командный интерпретатор) по умолчанию для текущего пользователя
$PS1, **$PS2**	Формат «приглашения» командной строки

и т.д.

Для манипулирования набором переменных служат команды **`set`** и **`unset`**:

`set` – *вывод всех переменные окружения*

`set` ***`список_значений`*** – *инициализация позиционных параметров значениями из списка*

`set \`ls\`` – *передача в позиционные параметры результатов выполнения команды* **`ls`**

`unset` ***`имя_переменной`*** – *прекращение действия переменной*

Объявленные переменные, в отличие от унаследованного окружения, не передаются порожденным процессам. Чтобы обеспечить такую передачу, используется команда **`export`**:

```
X=123
...
export X
```

Экспортироваться должна переменная по имени, а не ее значение **`$X`**!

В любом случае, порожденным процессам переменные передаются исключительно ***по значению***. Изменения в окружении процесса-потомка не влияют на процесс-родитель.

Ряд переменных являются ***предопределенными*** (***параметрами***)

Позиционные параметры обеспечивают доступ к аргументам командной строки (аналог – массив **`argv[]`** в программах на C):

`$0` – «имя программы» (аналогично программам на C); в случае скрипта это имя файла скрипта, если интерпретатор выполняет файл, или имя интерпретатора, если он выполняет входной поток

`$1`, **`$2`**, …, **`$9`** – остальные аргументы командной строки

Доступ к следующим (10, 11 и т.д.) позиционным параметрам – посредством команды **`shift`**

Количество актуальных позиционных параметров – параметр **`$#`** (аналог – **`argc`** в программах на C).

Некоторые другие (***специальные***) параметры:

Имя	Интерпретация
`$*`	все позиционные параметры командной строки, начиная с 1-го, как единая строка с разделителями
`$@`	то же, но в виде списка строк
`$#`	количество позиционных параметров без учета 0-го (`$0`)
`$$`	PID текущей программы; в случае скрипта это PID выполняющего его интерпретатора shell
`$?`	код завершения последней не-фоновой команды (конвейера)
`$!`	PID последней запущенной фоновой команды (конвейера)

и т.д.

Содержимое командной строки может быть изменено самим скриптом.

В частности, для многих интерпретаторов прямой доступ возможен лишь к первым 9 позиционным параметрам, поэтому их список приходится сдвигать командой **`shift`**. При этом «выдвинутые» значения теряются, и корректируются значения **`$#`**, **`$*`** и **`$@`**.

Сдвиг на один шаг влево, т.е. **`$1`**←**`$2`**, **`$2`**←**`$3`** и т.д.:

```
shift
```

Сдвиг влево на заданное число шагов:

```
shift N
```

Некоторые простейшие операции над переменными (наличие операций, выполняемых с учетом условий, позволяет во многих случаях избежать гораздо более громоздких конструкций ветвления или выбора):

`${var}string` – *простое объединение (катенация) значения переменной* **`$var`** *со строкой* **`string`**

`${var:-string}` – *значение* **`$var`** *если оно определено и не пусто, иначе строка* **`string`**

`${var:=string}` – *аналогично, но значение* **`$var`** *также заменяется на* **`string`**

`${var:+string}` – *строка* **`string`** *если переменная* **`var`** *определена и не пуста*

`${var+string}` – *строка* **`string`** *если переменная* **`var`** *определена*

${*var*:?*message*}** – *вывод сообщения* ***message*** *и завершение интерпретатора, если значение* ***$var* *пусто или не определено*

Например:

```
${Critical_Var:?"Critical variable is not set!"}
```

4.5 Управляющие конструкции (операторы)

4.5.1 Ветвление по условию

```
if команада_условие
then
  команды_если_true
else
  команды_если_false
fi
```

Проверяется ***код возврата*** команды-условия.

Ветвь «**`else`**» может отсутствовать, но завершающее ключевое слово «**`fi`**» необходимо.

Более компактная запись за счет использования разделителя «;»:

```
if команада_условие ; then ; команда1 ; команда2
; … ;
else ; команда11 ; команда12 ; … ; fi
```

Сокращенная запись для вложенных ветвлений:

```
if команада_условие1
then
  …
elif команада_условие2
  …
  else
    …
fi
```

4.5.2 Переключатель (выбор *варианта)*

```
case $переменная in
вариант1) команды1 ;;
вариант2) команды2 ;;
…
*) команды_варианта_по_умолчанию
esac
```

Проверяемые варианты – регулярные выражения, с которыми сравнивается заданное строковое значение (обычно значение переменной)

4.5.3 Циклы по условию

С прямым условием («выполнять пока TRUE, т.е. =0»)

```
while команада_условие
do
  команды_пока_true
done
```

С инверсным условием («выполнять пока FALSE, т.е. ≠0»)

```
until команада_условие
do
  команды_пока_false
done
```

Компактная форма аналогично ветвлению:

```
while команада_условие ; do ; команда1 ; команда2
; … ; done
```

4.5.4 Цикл-итератор

```
for переменная_цикла in список_значений
do
   команды_цикла
done
```

В частности:

```
for par in $@ – цикл по всем позиционным
параметрам
…
for file_name in `ls` – цикл по всем именам
файлов текущего директория
…
for file_name in * – тоже цикл по всем именам
файлов текущего директория
…
```

Переменная цикла сохраняется за пределами цикла, в ней остается последнее присвоенное значение.

4.5.5 Прочие

Принудительный безусловный выход из цикла:

```
break
```

Принудительный переход на следующую итерацию цикла:

```
continue
```

Выход, завершение текущего сценария или сеанса shell:

```
exit
```

4.5.6 Команда для выработки условий

Команда для выработки ***условий***:

`test` ***`arg1 arg2 arg3`*** …

Сокращенная форма (используется чаще полной):

`[` ***`arg1 arg2 arg3`*** … **`]`**

Разделители при квадратных скобках «**`[`** … **`]`**» необходимы!
Аналогично, разделители между аргументами.

Аргументы:
– опции – задают выполняемые операции
– параметры – операнды, над которыми они выполняются

Результат в любом случае – код возврата:

```
[ $1 == "abc" ]
echo '$1 == "abc"' $?
```

Команда поддерживает сравнения различного вида, группировку, логические операции:

Файлы	
`-f` ***`файл`***	файл существует, является регулярным (обычным)
`-s` ***`файл`***	файл существует и не пуст
`-d` ***`файл`***	файл является директорием
`-c` ***`файл`***	файл является символьным устройством
`-r` ***`файл`***	файл доступен для чтения
`-w` ***`файл`***	файл доступен для записи
`-x` ***`файл`***	файл доступен для выполнения

Строки	
`-z $s`	строка **`$s`** пуста (нулевой длины)

`-n $s`	строка `$s` не пуста (ненулевой длины)
`$s1 = $s2`	строки `$s1` и `$s2` равны
`$s1 != $s2`	строки `$s1` и `$s2` не равны

Целые числа	

Логические операции и группировки	

(параметры)

При выполнении операций сравнения (проверок) критично соответствие содержимого операндов ожидаемому проверкой.

(примеры)

4.6 Вычисление выражений (арифметических и логических)

Внутренняя команда **`expr`** («классический» shell **`sh`**):

`expr` ***`arg1 arg2 arg3`*** …

Список аргументов интерпретируется как вычисляемое выражение с синтаксисом, близким к привычному «математическому». Необходимы разделители между аргументами!
Аргументы: опции – выполняемые операции, параметры – операнды, также предусмотрены скобки для группировки.

Результат выполнения – вычисленное значение в потоке вывода, оно может быть перехвачено обычным образом.

Код возврата команды **`expr`**:

Значение		**Интерпретация**
числовое	**логическое**	
`0`	**`TRUE`**	результат непустой и не равен `0`
`1`	**`FALSE`**	результат пустой или равен `0`
`2` или `3`	**`FALSE`**	произошла ошибка

Поддерживаемые операции:

`+`	суммирование
`-`	вычитание
`\*`	умножение («`*`» – метасимвол, требует экрана)
`/`	деление
`%`	остаток от деления
`**`	возведение в степень (**`bash 2.02`** и выше)

(примеры)

Внутренняя команда **`let`** (bash):

`let` ***`arg1 arg2 arg3`*** **`...`**

Аргументы – вычисляемые выражения.
Выражения в целом подобны используемым командой **`expr`**, но имеют отличия в синтаксисе и шире по возможностям.

Внутреннее представление при вычислениях – целые числа фиксированной разрядности, без контроля переполнения, но с перехватом попытки деления на ноль.

Результат выполнения – модификация участвующих в выражениях переменных, без вывода данных в поток вывода

Код возврата команды **`let`**:

Значение		**Интерпретация**
числовое	**логическое**	
0	**TRUE**	результат =0
1	**FALSE**	результат ≠0

(Таблица – действия)

Примеры:

```
x=1; let y=x+1
let "x += 1"
```

«Составная» команда **((…))** (bash):

((*выражение*))

(Строго говоря, «**((**» и «**))**» – операторы, но их рассматривают практически только в составе этой конструкции.)
Эту форму обычно считают более предпочтительной, чем **let**

Пример:

```
x=1; (( x++ )); y=$(( x+1 ))
```

4.7 Ввод-вывод, взаимодействие с другими процессами и с системой

4.7.1 Вывод (в stdout):

`echo` ***`список выводимых данных`***

Вывод в **`stderr`**, в другой поток, файл или устройство – используя перенаправление **>**, **>>** и обращения к потокам.

примеры

4.7.2 Явный ввод (из stdin):

`read` *`имя_переменной`*

пример

Обращение к переменной – по имени, а не к значению!
В противном случае значение указанной переменной будет интерпретировано как имя вводимой переменной (скорее всего, несуществующей). Это можно использовать для «переключаемого» ввода:

пример

Ввод из другого потока, файла, устройства – используя перенаправление **<** и обращения к потокам

пример

4.7.3 Выполнение внешней команды

Обычно внешние команды выполняются либо как процесс-потомок, порождаемый из двоичного файла, либо как процесс – интерпретатор скрипта (тоже потомок).

Выполнение команды с полным перекрытием (замещением) текущего shell:

```
exec командная строка
```

4.7.4 Перехват и обработка сигналов

`trap "`*`команда1 команда2 ...`*`" №сигнала1 №сигнала2 ...`

Обращения к всем доступным внешним программам в системе как к командам.

4.8 Функции

Поддерживаются в **bash** и ряде других современных интерпретаторов.
Не поддерживаются в классическом **sh**.

Функция – альтернатива вызываемому внешнему файлу-скрипту. Синтаксис вызова функции и порядок передачи ей аргументов подобны обычному вызову команды.

Объявление функции:

```
function имя_функции () список_команд
```

Список команд (составная команда) обычно заключается в фигурные скобки **{ … }** – выполнение в текущем экземпляре интерпретатора. Использование круглых скобок – выполнение в новом экземпляре – возможно, но обычно не представляет интереса.
Ключевое слово «**function**» не обязательно, обычно оно опускается.

Вызов функции:

имя_функции арг1 арг2 …

Примеры функций и их использования:

```
Add ()
{
  expr $1 + $2
}
…
x=2; y=3
echo "x+y =" `Add $x $y`
```

```
Pow ()
{
 result=1 ; i=$2
 while [ $i -ge 1 ] ; do
    result=`expr $result \* $1`
    i=`expr $i - 1`
 done
 echo $result
}
…
x=2; y=3
echo "x^y = " `Pow $x $y`
```

4.9 Прочие возможности

Некоторые комментарии «специального назначения»:

Явное указание интерпретатора для выполнение скрипта, применяется текущим интерпретатором при обращении к файлу как к исполняемому:

!# *интерпретатор*

Эффективный механизм управления выполнением разнородных скриптов, но осложнен перенос в другую систему, т.к. необходимо указывать точным абсолютный путь к интерпретатору (поиск не предусмотрен). Разделитель после маркера не обязателен. Например:

```
!# /bin/bash
... #текст скрипта bash
```

```
!#/usr/bin/perl
... #текст программы perl
```

Описание файла для команды **what** (присутствие в системе опционально):

```
#@(#)описание
```

В действительности команда **what** ориентируется на строку-маркер «**@(#)**». Такое описание можно включить не только в текст (скрипт), но и в двоичный файл.